White Paper

Intel Software Solutions Group

Yongnian Le

Schema Validation with Intel® Streaming SIMD Extensions 4 (Intel® SSE4)

Intel® SSE4 is a new set of Single Instruction Multiple Data (SIMD) instructions that improve the performance and energy efficiency of a broad range of applications. Intel® SSE4.2 is a new subset of Intel® SSE4 instructions that will be introduced in the 45nm Next Generation Intel® Core™2 processor family.

This white paper describes how schema validation can utilize Intel SSE4.2 instructions to achieve great performance speedups and memory consumption optimization in attribute/element declaration lookup, string enumeration facet validation, and model group state transion, three hot bottlenecks in validation runtime.

April 2008

Intel Corporation

Contents

Figures

Tables

Introduction

Intel® Streaming SIMD Extensions 4 (Intel® SSE4) is a new set of Single Instruction Multiple Data (SIMD) instructions designed to improve the performance of various applications, such as video encoders, image processing, 3D games, and string/text processing. Intel SSE4 builds upon the Intel® 64 and IA-32 instruction set, the most popular and broadly used computer architecture for developing 32-bit and 64-bit applications. Intel SSE4.2 is a new subset of Intel SSE4 instructions that will be introduced in the 45nm Next Generation Intel® Core™2 processor family.

This white paper will describe how schema validation can benefit from the Intel SSE4.2 instructions, achieving great performance speedups and memory consumption optimization in attribute/element declaration lookup, string enumeration facet validation, and model group state transion, three hot bottlenecks in validation runtime.

Parallel TRIE w/STring and Text Processing Instructions

It is the common and hot bottleneck during schema validation to search and select one item from a set of strings, especially when the string is pretty long. More specifically, the operation involves:

- In cases containing string enumeration, select an input string from a set of validate strings.
- Look up attribute declaration according to attribute namespace and local name.
- Look up element declaration according to element namespace and local name.
- Look up next state number in model group DFA according to current state number, element namespace, and local name.

The quality of the validation is a factor that determines the comparison speed it takes. This string lookup operation is often the target of performance optimization to improve the schema validation quality.

In previous implementation, schema validation took different algorithms and data structures handling above search and selection request.

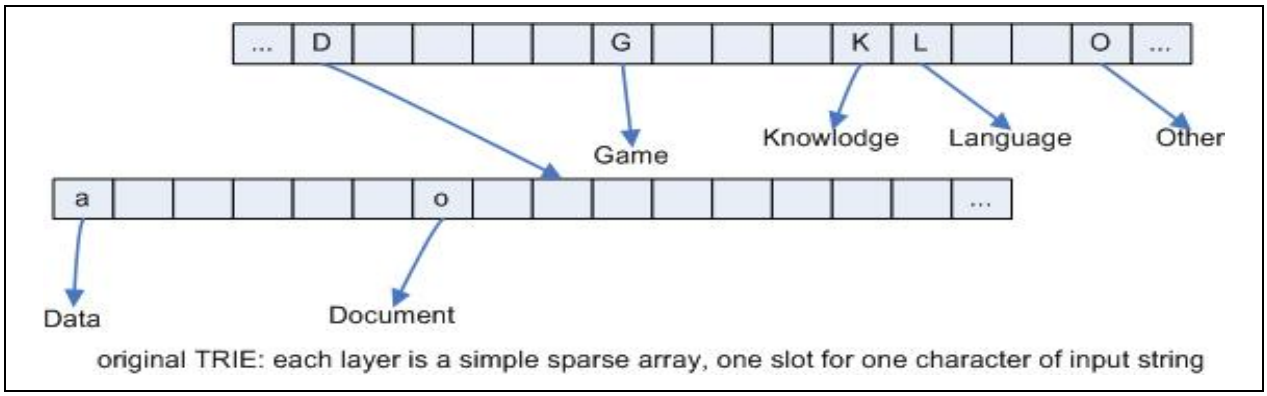
- Convert string to a unified number according to string content by hash table and lookup string from string set by another hash table lookup or array linear search. The combined operations often account for about 30%-35% of the total CPU cycles consumed by different validation event handlers of a SAX style schema validation, especially when the string is pretty long.
- Use two-dimensional binary search algorithm to look up string from a pre-sorted string set. It often accounts for about 15%-20% of the total CPU cycles consumed by different validation event handlers of a SAX style schema validation, on such kind of benchmark.

The Intel SSE4 instruction, PCMPESTRI/PCMPESTRM/PCMPISTRI/PCMPISTRM, as String and Text Processing Instructions, are widely used by developers to optimize application performance for string comparison scenarios. Those instructions compare two strings in parallel by a rich set of programmable modes with an immediate byte as programming

control[1]. By utilization those string and text processing instructions, a string dictionary by parallel TRIE data structure is invented for efficient string lookup while with much smaller memory consumption.

For traditional TRIE organization, shown by Figure 1, each TRIE layer is a simple 256-slot sparse character array, assuming that all characters are within ASCII range. According to current character of input string, next layer TRIE could be found quickly by "NEXT(string, position) = sparse_array[string[position]]". In the real practice, strings in the TRIE often share a lot of common prefix or characters so that there are few slots of the array filled with meaningful pointer and it always takes several lookups between layers to finally locate the associated string.

Figure 1. Original TRIE Organization



The Parallel TRIE overcomes above weakness completely and proposes to (1) organize each TRIE layer from big sparse array to *compact string*; (2) combine adjacent two characters (bytes) as one *short atomic integer* to reduce likelihood of collision so that less TRIE layer is needed; (3) organize parallel TRIE to fully utilize parallel string comparison capability provided by Intel SSE4.2/STTNI for high performance string query operation. Table 1 shows the difference of parallel TRIE compared with traditional TRIE.

Table 1. Parallel TRIE w/ STring and Text Processing New Instructions (STTNI)

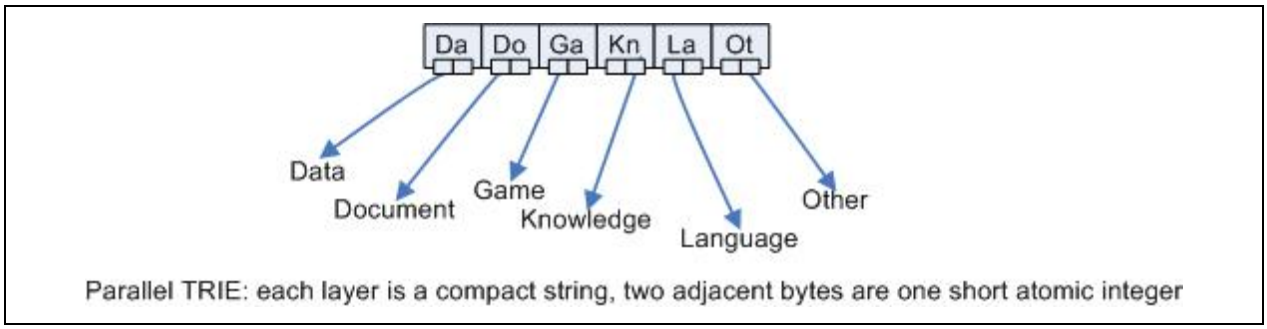
Difference Compared with Traditional TRIE	Advantages Compared with Traditional TRIE	How String and Text Processing Instructions are Used
Use two bytes (16-bit short integer) as one atomic unit.	Conflict rate is reduced.	Use word (16-bit short integer) granularity mode of string and text processing instructions.
Each parallel TRIE layer's data structure changes from sparse array to string.	Small memory usage with relatively fast access with the help of new instructions.	Use string and text processing instructions to find atomic unit from the each parallel TRIE layer.
String lookup can utilize 16 bytes of input string at a time.	Quickly locate string in the parallel TRIE with the help of new instructions.	Use string and text processing instructions to effectively compare the strings, especially long strings.

1 For more details on String and Text Processing Instructions, see the *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2B and Intel® SSE4 Programming Reference*.

Figure 2 shows the example of parallel TRIE layer organization. For each slot in parallel TRIE layer, there is a link pair serving for two different purposes:

- When there is no next level parallel TRIE data structure, one link pointing at odd length string, and the other link pointing at even length string.
- When there is next level parallel TRIE data structure, one link pointing at next level TRIE data structure for those strings, which have another 16 bytes to look up, the other link pointing at other strings, which have no more 16 bytes to look up.

Figure 2. Example of Parallel TRIE Organization



In parallel TRIE, most of the lookup between layers could be reduced and parallel string comparison could be achieved by careful selection of short atomic integer from input string. When selecting short atomic integer of input string, different from traditional TRIE which directly uses the character at current string position, parallel TRIE uses the non-conflicting short integer within the *selection window* (from current position to 16 bytes more) of the input string. Here *conflict* means that there is one short integer within the selection window, which is already in the compact string of current parallel TRIE layer. If conflict happens, another short integer is chosen from the selection window and is inserted just before the conflicting position. If there is no short integer available to resolve conflict, such string will either be added to fail through strings, the number of which is generally small and appropriate data structure like list or hash table could be chosen, or be constructed as next parallel TRIE, depending on whether input string is long enough (having another 16 bytes) for another lookup.

Figure 3. Parallel TRIE Construction by Appropriate Atomic Short Integer Selection

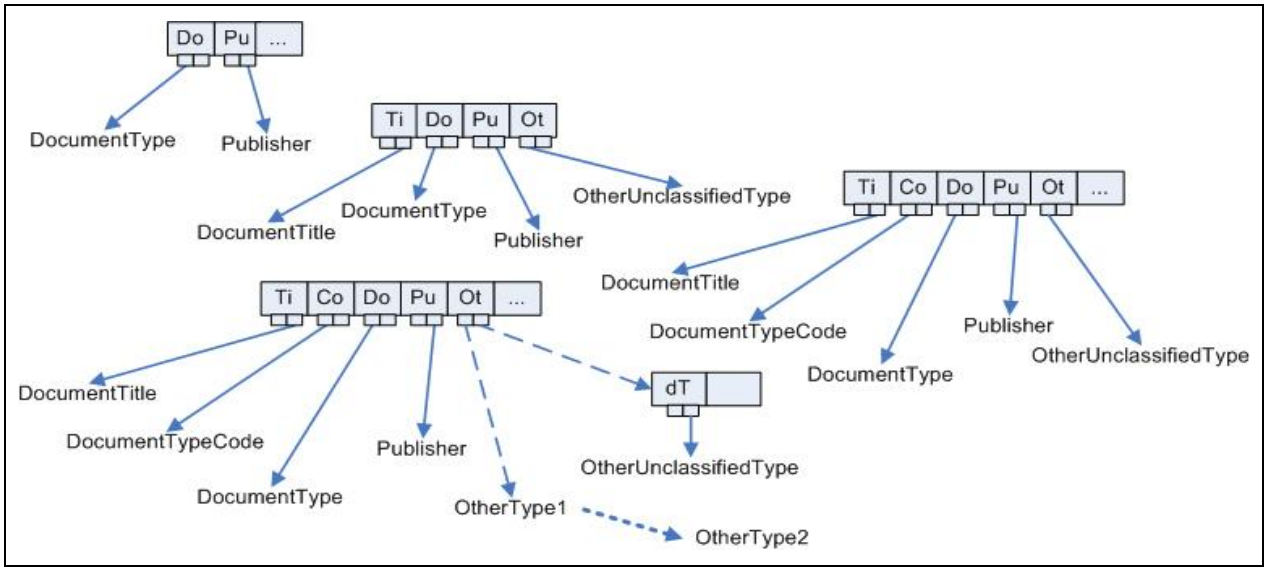
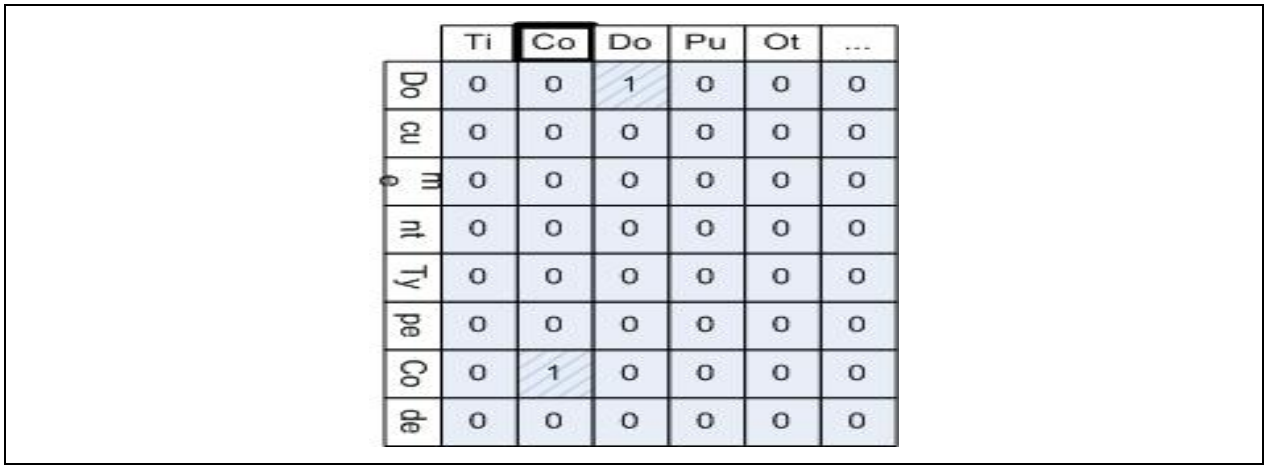


Figure 3 demonstrates the process of how parallel TRIE is constructed step by step. Assuming “DocumentTypeCode” is to be inserted after “DocumentType”, short integer “Do” causes conflict because it is already in the compact string after “DocumentType” is inserted. Therefore, another short integer “Co” is chosen from the selection window ranging from ‘D’ to ‘e’ and gets inserted just before “Do”. After “OtherUnclassifiedType”, “OneType1” and “OtherType2” are inserted one by one, “OtherType1” and “OtherType2” form the fail through strings and “OtherUnclassifiedType” forms another parallel TRIE layer.

When input string is queried on the constructed parallel TRIE, the selection window of such input string is compared with the compact string of parallel TRIE. The left most satisfied slot is selected for further action, depending on whether it points at a fail through strings, another parallel TRIE or just associated string. If it doesn’t point at associated string, shown as dash line in Figure 3, it could be either fail through strings which have no more 16 bytes to look up or another parallel TRIE data structure which needs recursive lookup after shifting selection window by 16 bytes. Finally, a string comparison is needed to verify whether the located associated string is the same as input string.

With the parallel string comparison capability provided by Intel SSE4.2/STTNI techniques, the string lookup could be finished in dramatically efficient parallel manner: comparing 16 bytes from selection window of input string and 16 bytes from the compact string of parallel TRIE layer by just one hardware SIMD instruction. Figure 4 shows how hardware SIMD instruction on Intel Nehalem platform looks up string “DocumentTypeCode” in the parallel TRIE of Figure 3: it compares “DocumentTypeCode” and “TiCoDoPuOt” by one SIMD instruction and the left most satisfied slot “Co” is found.

Figure 4. Dramatically Accelerate String Lookup by Intel® SSE4.2/STTNI



Intrinsic function `_mm_cmpestri` which wraps the Intel SSE4.2 instruction PCMPESTRI is provided by Intel® C/C++ Compiler 10.0. A constant mask is set to let the instruction compare input string and rule string as 16-bit short integer unit and return the left-most index of matched rules in parallel. When there is no hardware Intel SSE4.2/STTNI support in other platforms or compiler cannot recognize such intrinsic function, a software-optimized (inline and immediately return when first matched rule found) simulation function is provided as well to perform the same functionality.

Figure 5 shows the code fragment of string lookup to find next level information in parallel TRIE data structure, and Figure 5 shows the code fragment of string lookup to resolve conflict in selection window.

Figure 5. Code Fragment of String Lookup to Find Next Level Information

```
#define STTNI_MASK          SIDD_CMP_EQUAL_ANY | SIDD_UWORD_OPS |
SIDD_LEAST_SIGNIFICANT
unsigned short * rules = curr_trie->getRules();
unsigned short * input = static_cast<unsigned
short*>(static_cast<void*>(const_cast<char*>(string + position)));
unsigned short * pt = rules;
int index = 8, len = next_size;
#if defined(_STTNI)
if (usingSTTNI) {
    const __m128i ahead = _mm_loadu_si128((__m128i*)input);
    while (len > 0 && (index = _mm_cmpestri(ahead, actlen,
_mm_loadu_si128((__m128i*)pt), len, STTNI_MASK)) == 8) {
        pt += 8; len -= 8;
    }
}
else
#endif
{
    while (len > 0 && (index = pcompstri_eqany(input, (actlen > 8? 8 : actlen), pt,
(len > 8? 8 : len))) == 8) {
        pt += 8; len -= 8;
    }
}
if (index == 8) return NULL;
index += (pt - rules); //index is the what we need
```

Figure 6. Code Fragment of String Lookup to Resolve Conflict in Selection Window

```
#define STTNI_MASK          SIDD_CMP_EQUAL_ANY | SIDD_UWORD_OPS |
SIDD_LEAST_SIGNIFICANT
for (int m = 0; m < size; m++) {
Item listitem = static_cast<Item>(list_trie->getNextLink(m));
    int oldlength = Helper::length(listitem), actrulelen = oldlength & (~0x1);
    actrulelen = actrulelen - position > 16? 8 : (actrulelen - position) >> 1;
    unsigned short * rules = static_cast<unsigned short*>(Helper::string(listitem) +
position);
#if defined(_STTNI)
    if (usingSTTNI) {
        mask |= (int)_mm_cvtsi128_si32(_mm_cmpestrm(_mm_loadu_si128((__m128i*)rules),
actrulelen, _mm_loadu_si128((__m128i*)input), actinputlen, STTNI_MASK));
    }
    else
#endif
    {
        mask |= (int)pcompstrm_eqany(rules, actrulelen, input, actinputlen);
    }
}
for (int position = 0; position < actinputlen; position++) {
    if ((mask & (1<<position)) == 0) {
        nextrule = input[position]; // next rule is what we need
        break;
    }
}
```

Results

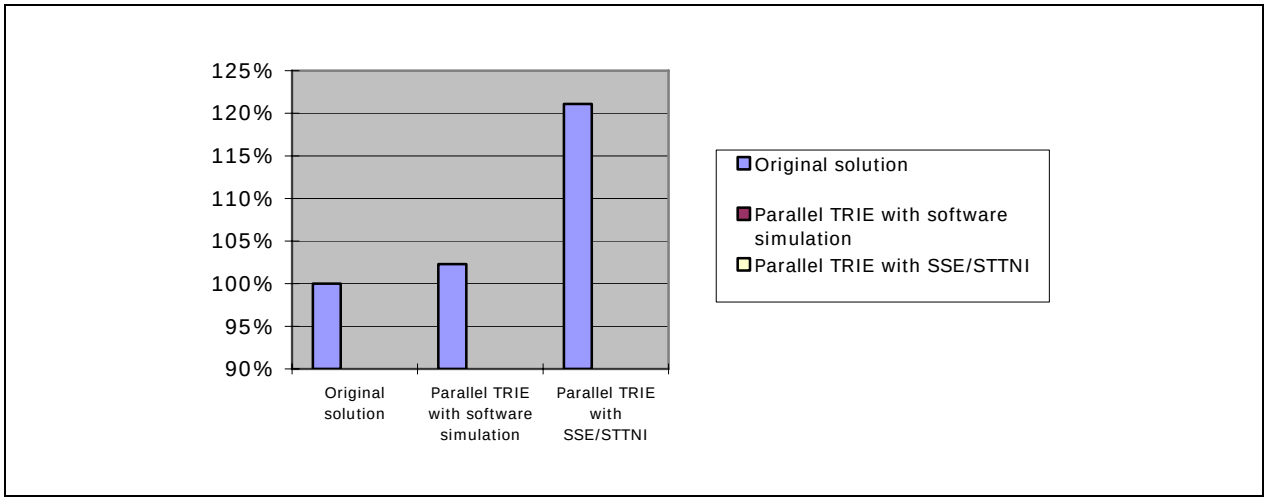
Schema validation processing consists of two components: SAX parsing and validation runtime processing by a series of call back handlers. Each of them occupies almost half of the CPU clock ticks of the overall schema validation according to the profiling data collected by VTune™ analyzer.

In schema validation call back functions, below hot bottlenecks where schema validation needs look up associated information according to the input string are replaced with parallel TRIE data structure. When there is no Intel SSE4.2/STTNI instruction available, a software-optimized (inline and immediately return when first matched rule found) simulation function is provided as well to perform the same functionality.

- In cases containing string enumeration, select an input string from a set of validate strings.
- Look up attribute declaration according to attribute namespace and local name.
- Look up element declaration according to element namespace and local name.
- Look up next state number in model group DFA according to current state number, element namespace and local name.

Figure 7 shows overall schema validation speed-up where all above hot bottlenecks are replaced by parallel TRIE data structure. The performance data is collected from Nehalem platform with 1G memory and there are 14 schema validation cases for benchmarking. If there is no hardware Intel SSE4.2/STTNI support, there is minor improvements, but if there is hardware Intel SSE4.2/STTNI support on Nehalem platform, there could be averagely about 21% performance speed up.

Figure 7. Schema Validation Speed Up by Parallel TRIE Enhancement



Conclusion

The use of Intel SSE4 instructions was shown to improve both performance and memory consumption in attribute/element declaration lookup, string enumeration facet validation, and model group state transion, three hot bottlenecks in validation runtime.